AIM Technical Note #58

Notes from Charlie Golvin

Charlie Golvin November 1, 1990

This technical note is an informal compendium of issues and techniques of interest to CD-I developers. It includes discussion of non real-time sectors in real-time files, the use of error detection code in Form 2 sectors, the use of AIM "include" files, the portation of code to the SUN SPARC environment, the use of SCCS, error detection in PCLs, and two useful programs

Notes from Charlie Golvin

1. Non Real-Time Sectors in Real-Time Files

Recently we have received clarification from Philips in Eindhoven regarding some discrepancies in the *CD-I Full Functional Specification* (also known as the Green Book). This clarification specifies that it is legal to place non real-time sectors (that is, sectors with the real-time bit cleared) in a real-time file. The biggest advantage of this capability is that it makes it possible to receive error-corrected sectors asynchronously from a real-time file. Previously, the only method for the delivery of error-corrected sectors was out of a non real-time file using the synchronous read() call.

As a general strategy, I recommend the placement of critical data in Form 1 non real-time data sectors immediately preceding the real-time data for which this critical data is required. These non real-time sectors could be terminated with an end of record bit if it is necessary to deliver these data sectors without continuing into the following data.

Some caveats regarding the use of these sectors:

1) Do not interleave these sectors with sectors that depend on the real-time delivery of data, especially audio data. If the driver is forced to perform error correction, the remaining sectors will not be delivered in real-time, irrespective of the real-time bit in those sectors.

2) *DBL*, AIM's Disc Building Language, does not currently support this syntax. It automatically places data sectors into Form 1 real-time sectors. You will have to edit the .rrb script produced by *DBL* to clear the real-time bits in the data sectors to be error-protected.

3) While the disc builder bd does support the placement of non real-time sectors in a real-time file, *Mac CD-I* does not. We hope to receive an updated version of *Mac CD-I* in the near future.

4) Only the newest version of the CD/AP driver supports the delivery of error-corrected sectors from within a real-time file. You will need either edition 6 (or newer) of *cdapdriv* or the C ROMs from AIM to actually test an implementation of this strategy. If you do not have either this software or these ROMs, please contact Charles Golvin at AIM.

2. Error Detection Code in Form 2 Sectors

For those using the ISG disc builder bd (either on a Sun or a Macintosh), these tools support the placement of an EDC code in the final four bytes ("quality control bytes") of Form 2 sectors. When making final discs for mastering, we recommend that you use this feature. To take advantage of this feature, add the following line to your map file:

```
SET EDC ON.
```

3. Using the AIM-Supplied Include Files

For those using the include files supplied by AIM for the purpose of version control, these files have SCCS identifier strings embedded in them as static strings. To tell the compiler not to include these strings in your final executable, define the symbols lint and NO_BVC to the C preprocessor. On the Microware cross-compiler, this is achieved by adding the options -dlint -dNO_BVC to the compile line.

You should also be aware that shortly, with the release of version 1.1 of CD-RTOS, we will be reverting to the original Microware-provided include files. Changes that we have made to these files will be separated into discrete AIM-specific include files.

4. Porting Code to SPARC Architecture

The primary difference between Sun3 and SPARC machines is the alignment of data types: on Sun3 machines all words and longwords fall on word boundaries; on SPARC machines words fall on word boundaries and longwords (ints) must fall on longword boundaries. For structures that are used purely for internal purposes, this does not cause problems. It is only in the case of attempting to read data off disk (or other external device) directly into such a structure where your program can get into trouble.

As a general rule, when designing structures that may need to cross machine boundaries to SPARC architecture, you should organize a structure's members by grouping the same data types together in order of decreasing size. Thus, I first list all the pointers and ints (i.e., four byte quantities), followed by all the shorts (two byte quantities), and then all the byte quantities. The Sun C compiler still pads the structure to the nearest longword (four bytes), but does not introduce extra padding between members of the structure.

What do you do when your existing structures and working code do not follow these guidelines? The easiest method, although it is on the ugly side, is to introduce new structures that allow you to read data from disk into memory. Then you can transfer data between this interim structure and your old definition so you do not

have to modify a lot of code. This way you modify only the code that does the input from and output to disk. Perhaps this is best explained with an example. Suppose I had a structure definition and variable declaration as follows:

```
typedef  struct  _poor_choice
{
     short  eeny;
     int  meeny;
     short  miney,  moe;
}POOR_CHOICE;

POOR_CHOICE  whoops;
```

On a Sun3, sizeof(whoops) is ten, and, on a SPARC, it works out to twelve because of the two pad bytes the compiler must introduce between **eeny** and **meeny** to make **meeny** fall on a longword boundary. For the purposes of reading and writing these structures, I make the following structure definition and variable declaration:

```
typedef  struct  _better_choice
{
     short  eeny;
     short  high_meeny;
     short  low_meeny;
     short  miney,  moe;
}BETTER_CHOICE;

BETTER_CHOICE  workaround;
```

Then, having read the data for **workaround** into memory, I copy it to **whoops** as follows:

```
whoops.eeny  =  workaround.eeny;
whoops.meeny  =  (int)(workaround.high_meeny  <<  16)  +
workaround.low_meeny;
whoops.miney  =  workaround.miney;
whoops.moe  =  workaround.moe;
```

Thus, I do not have to change all the interior code I have for dealing with the various members of structures of type POOR_CHOICE at the cost of a new typedef. I play the same type of game (in reverse, of course) for outputting something of type POOR_CHOICE.

5) How to Use SCCS

Please see the attached note to get a basic understanding of what you need to know to use *sccs*, the Source Code Control System, on the Sun.

6) Error Detection in PCLs

The Green Book allows machine developers to support error detection information that is either byte or word resolution. The resolution determines how much memory your application must provide to catch errors during a play. That is, in order to flag errors on a machine with a CD driver with byte resolution, your error block buffer needs to be twice as large as that for a machine with a CD driver with word resolution. While there is a field in the PL_ERR data structure that defines the resolution of the particular machine's driver, some uncertainty has existed as to how to retrieve this information. The answer is to create a PL_ERR structure, attach it to a PCL used during a play, wait for an error to occur, and then examine the Err_Res field of the PL_ERR structure. While this is not as satisfying an answer as, for example, to query the CSD, the justification is that an application needs to make the provision for error detection irrespective of the error resolution of a machine. Thus, your application should be able to allocate enough memory to catch errors on a machine that has byte resolution. Following an error, if it discovers that the machine is capable of word resolution, it simply has more memory to devote to other endeavors.

The bottom line on this point is that for your application to be able to recover from errors in what it considers critical data, it must be able to supply enough memory to handle the case of a CD driver that is capable only of byte resolution. To refresh your memory, this is on the order of one bit per byte of data to be error checked: 294 bytes for a Form two sector, 256 bytes for a Form one sector.

7) Two Useful Programs

I have created two programs that are useful for testing the behavior of your title in one megabyte without having to pull the memory board every time you want to do this. The two programs are called **nosysram** and **cload**. I recommend that you copy these two programs into the CMDS directory on your player's boot floppy. The following explains the use of the programs:

nosysram: This program is a "memory gobbler." It simply allocates all of system RAM and then goes to sleep. Of course, when **nosysram** exits, all system RAM returns, so you will want to run **nosysram** in the background before starting your program. Because it goes to sleep as soon as it finishes grabbing memory, it will not consume any CPU.

cload: This program is essentially like the OS-9 utility **load**, except that **cload** loads the desired module into the specified plane of colored memory. You specify the plane by use of the -p option. The default plane is plane A. For example, both of the following commands result in **myprogram** being loaded into plane A:

```
cload  /dl/myprogram
cload  -pa  /dl/myprogram
```

and

```
cload  -pb  /dl/myprogram
```

results in myprogram being loaded into plane B.

Thus, to emulate the behavior of a one megabyte player, perform the following steps (assuming that both nosysram and cload exist in /dd/cmds and that these steps immediately follow chaining to CD-RTOS from the player shell):

```
load  nosysram
load  cload
nosysram  &
cload  -pb  myprogram
myprogram
```

These programs are available from AIM to AIM co-producers and may also be found on the Engineering 6.0 disc. Contact Charles Golvin.